yarp Documentation

Release 0.4.0

Jonathan Heathcote

Dec 24, 2018

Contents:

This library facilitates a programming style which is a little bit like (functional-ish) reactive programming.

CHAPTER 1

Introduction

This programming style will be familiar to anyone who has used a spreadsheet. In a spreadsheet you can put values into cells. You can also put functions into cells which compute new values based on the values in other cells. The neat feature of spreadsheets is that if you change the value in a cell, any other cell whose value depends on it is automatically recomputed.

Using `yarp` you can define `Value`s, and functions acting on those values, which are automatically reevaluated when changed. For example:

```
>>> from yarp import Value, fn

>>> # Lets define two Values which for the moment will just be '1'
>>> a = Value(1)
>>> b = Value(1)

>>> # Lets define a function 'add' which adds two numbers together. The
>>> # @fn decorator automatically wraps 'add' so that it takes Value
>>> # objects as arguments and returns a Value object. Your definition,
>>> # however, is written just like you'd write any normal function:
>>> # accepting and returning regular Python types in boring every-day
>>> # ways.
>>> @fn
... def add(a, b):
...     return a + b

>>> # Calling 'add' on our 'a' and 'b' Value objects returns a new Value
>>> # object with the result. Get the actual value using the 'value'
>>> # property.
>>> a_plus_b = add(a, b)
>>> a_plus_b.value
2

>>> # Changing one of the input values will cause 'add' to automatically be
>>> # reevaluated.
>>> a.value = 5
```

(continues on next page)

(continued from previous page)

```
>>> a_plus_b.value
6
>>> b.value = 10
>>> a_plus_b.value
15

>>> # Accessing attributes of a Value returns a Value-wrapped version of
>>> # that attribute, e.g.
>>> c = Value(complex(1, 2))
>>> r = c.real
>>> r.value
1
>>> i = c.imag
>>> i.value
2
>>> c.value = complex(10, 100)
>>> r.value
10
>>> i.value
100

>>> # You can also call (side-effect free) methods of Values to get a
>>> # Value-wrapped version of the result which updates when the Value
>>> # change:
>>> c2 = c.conjugate()
>>> c2.value
(10-100j)
>>> c.value = complex(123, 321)
>>> c2.value
(123-321)
```

As well as representing continuous values which change at defined points in time `yarp` can also represent values which are defined only instantaneously, for example an ephemeral sensor reading. For example:

```
>>> from yarp import Value, instantaneous_fn

>>> # Lets create an instantaneous value which occurs whenever a car drives
>>> # past a speed check. At the moment of measurement, the value has the
>>> # instantaneous value of the car's speed in MPH. For now, though, it
>>> # has no value.
>>> car_speed_mph = Value()

>>> # We live in a civilised world so lets convert that into KM/H. This
>>> # 'instantaneous_fn' decorator works just like the 'fn' one but returns
>>> # instantaneous values.
>>> @instantaneous_fn
... def mph_to_kph(mph):
...     return mph * 1.6

>>> car_speed_kph = mph_to_kph(car_speed_mph)

>>> # Lets setup a callback to print a car's speed whenever it is measured
>>> def on_car_measured(speed_kph):
...     print("A car passed at {} KM/H".format(speed_kph))
>>> car_speed_kph.on_value_changed(on_car_measured)
<function ...>
```

(continues on next page)

(continued from previous page)

```
>>> # Now lets instantaneously set the value as if a car has just gone past
>>> # and watch as our callback is called with the speed in KM/H
>>> car_speed_mph.set_instantaneous_value(30)
A car passed at 48.0 KM/H
```

As in these examples, the intention is that most `yarp`-using code will be based entirely on passing *`Value`*s around between functions wrapped with *`fn()`* and *`instantaneous_fn()`*.

CHAPTER 2

Creating impure Values

In general, `yarp` `Value`s are intended to be passed between simple, pure functions wrapped by the `fn()` decorator. Specifically, these functions don't hold any state and the resulting `Value`s change when-and-only-when any input `Value` changes. This type of function is very easy to write and reason about with `yarp` but is fundamentally constrained. For example, it is not possible to implement `delay()` using such a function since input :py;class:*Value* changes do not immediately result in the `Value` changing. Simillarly, the `no_repeat()` function also cannot be replicated since it doesn't always change its output `Value` when its input changes.

To get around this limitation it is necessary to manipulate `Value`s 'by hand'. Lets begin by seeing how `no_repeat()` is implemented.

The following pseudo code implementation goes the 'obvious' implementation for a no-repeat value:

```
on source value changed:
    if source value != last source value:
        output value = source value
    last source value = source value
```

The actual Python implementation looks like:

```
>>> def no_repeat(source_value):
...     last_value = source_value.value
...
...     # Initially take on the source value
...     output_value = Value(last_value)
...
...     @source_value.on_value_changed
...     def on_source_value_changed(new_value):
...         nonlocal last_value
...         if new_value != last_value:
...             last_value = new_value
...             # Copy to output whether continuous or instantaneous
...             output_value._value = source_value.value
...             output_value.set_instantaneous_value(new_value)
...
...     return output_value
```

In this example we create function (or rather, a closure) called `on_source_value_changed` and set it as the callback for the source *Value* using *Value.on_value_changed()*.

Note: This example uses the Python decorator syntax making the code read a little more naturally, as in the pseudo-code version.

The `last_value` variable is accessed from the enclosing scope is used to keep track of the last value received from the source. The nonlocal keyword is used to gain access to it from our callback.

The last detail is the way the output *Value* is updated. If `source_value` is a continuous function we could update the output using either:

```
output_value.value = new_value
```

Or:

```
output_value.value = source_value.value
```

However, if `source_value` is an instantaneous value, we'd need to do use *Value.set_instantaneous_value()*:

```
output_value.set_instantaneous_value(new_value)
```

Since we'd like to make our output *Value* mimic the input regardless of whether it is continuous or instantaneous, instead we use the following two-step process:

```
output_value._value = source_value.value
output_value.set_instantaneous_value(new_value)
```

By setting `_value` we change *Value.value* without triggering any callbacks registered with *Value.on_value_changed()*. We set this to the continuous value of the source (which is :py:data'NoValue' if the source is instantaneous). By calling *Value.set_instantaneous_value()* with the just-received value from the source we cause the callback to occur in the output *Value*.

You can try it out, first lets try a continuous value:

```
>>> # Create a value to de-repeat
>>> v = Value(123)
>>> nrv = no_repeat(v)
>>> nrv.on_value_changed(print)
<built-in function print>

>>> # Repeated values should not pass through
>>> v.value = 321
321
>>> v.value = 321
>>> v.value = 321
>>> v.value = 123
123
```

Next lets try an instantaneous value:

```
>>> # Create another instantaneous value to de-repeat
>>> iv = Value()
>>> nriv = no_repeat(iv)
>>> nriv.on_value_changed(print)
```

(continues on next page)

(continued from previous page)

```
<built-in function print>

>>> nriv.value is NoValue
True

>>> iv.set_instantaneous_value(123)
123
>>> nriv.value is NoValue
True

>>> iv.set_instantaneous_value(123)
>>> nriv.value is NoValue
True

>>> iv.set_instantaneous_value(321)
321
>>> nriv.value is NoValue
True
```

CHAPTER 3

`yarp` API

3.1 Value type

At the core of the `yarp` API is the *`Value`* type. This type is defined below.

`yarp.`**`NoValue = NoValue`**

A special value indicating that a `yarp` value has not been assigned a value.

`class` `yarp.`**`Value`**(*initial_value=NoValue*)

A continuous or instantaneous value which can be read and set.

This base class defines the fundamental type in `yarp`: the 'value'.

The actual data contained by this object should be regarded as immutable with changes being made by replacing the Python object with a new one to affect changes.

`value`

A property holding the current continuous value held by this object. If not yet set, or if this object represents only instantaneous values, this will be `NoValue`.

Setting this property sets the (continuous) contents of this value (raising the *`on_value_changed()`* callback afterwards).

To set the instantaneous value, see *`set_instantaneous_value()`*.

To change the value without raising a callback, set the `_value` attribute directly. This may be useful if you wish to make this Value mimic another by, in a callback function, setting `_value` in this Value directly from the other Value's *`value`* and calling *`set_instantaneous_value()`* with the passed variable explicitly. You must always be sure to call *`set_instantaneous_value()`* after changing `_value`.

`set_instantaneous_value`(*new_value*)

Set the instantaneous value of this Value, calling the on_value_changed callbacks with the passed value but not storing it in the *`value`* property (which will remain unchanged).

`on_value_changed`(*cb*)

Registers `callback` as a callback function to be called when this value changes.

The callback function will be called with a single argument: the value now held by this object. If the value is continuous, the value given as the argument will match the `Value.value` property. Otherwise, if this value is instantaneous, the value will not be reflected in the `Value.value` property.

Note: There is no way to remove callbacks. For the moment this is an intentional restriction: if this causes you difficulties this is a good sign what you're doing is 'serious' enough that `yarp` is not for you.

This function returns the callback passed to it making it possible to use it as a decorator if desired.

3.2 Aggregate Values

The `yarp` API provides a limited set of convenience functions which which turn certain native Python data structures into `Value`s which update whenever the underlying `Value`s do.

`yarp.`**`value_list`**`(`*list_of_values*`)`

Returns a `Value` consisting of a fixed list of other `Values`. The returned `Value` will change whenever one of its members does.

Parameters

list_of_values: [:py:class:'Value', . . .] A fixed list of `Value`s. The `value` of this object will be an array of the underlying values. Callbacks will be raised whenever a value in the list changes.

It is not possible to modify the list or set the contained values directly from this object.

For instantaneous list members, the instantaneous value will be present in the version of this list passed to registered callbacks but otherwise not retained. (Typically the instantaneous values will be represented by `NoValue` in `value` or in callbacks resulting from other `Value`s changing.

`yarp.`**`value_tuple`**`(`*tuple_of_values*`)`

A `Value` consisting of a tuple of other `Values`.

Parameters

tuple_of_values: (:py:class:'Value', . . .) A fixed tuple of `Value`s. The `value` of this object will be a tuple of the underlying values. Callbacks will be raised whenever a value in the tuple changes.

It is not possible to modify the tuple or set the contained values directly from this object.

For instantaneous tuple members, the instantaneous value will be present in the version of this tuple passed to registered callbacks but otherwise not retained. (Typically the instantaneous values will be represented by `NoValue` in `value` or in callbacks resulting from other `Value`s changing.

`yarp.`**`value_dict`**`(`*dict_of_values*`)`

A `Value` consisting of a dictionary where the values (but not keys) are `Values`.

Parameters

dict_of_values: {key: :py:class:'Value', . . . } A fixed dictionary of `Value`s. The `value` of this object will be a dictionary of the underlying values. Callbacks will be raised whenever a value in the dictionary changes.

It is not possible to modify the set of keys in the dictionary nor directly change the values of its elements from this object.

For instantaneous dictionary members, the instantaneous value will be present in the version of this dict passed to registered callbacks but otherwise not retained. (Typically the instantaneous values will be represented by `NoValue` in `value` or in callbacks resulting from other `Value`s changing.

3.3 Value casting

The following low-level funcitons are provided for creating and casting `Value` objects.

`yarp.`**`ensure_value`**`(`*value*`)`

Ensure a variable is a `Value` object, wrapping it accordingly if not.

- If already a `Value`, returns unmodified.
- If a list, tuple or dict, applies `ensure_value()` to all contained values and returns a `value_list`, `value_tuple` or `value_dict` respectively.
- If any other type, wraps the variable in a continous `Value` with the initial value set to the defined value.

`yarp.`**`make_instantaneous`**`(`*source_value*`)`

Make a persistent :py:class'Value' into an instantaneous one which 'fires' whenever the persistant value is changed.

`yarp.`**`make_persistent`**`(`*source_value*, *initial_value=NoValue*`)`

Make an instantaneous `Value` into a persistant one, keeping the old value between changes. Initially sets the `Value` to `initial_value`.

3.4 Value Operators

The `Value` class also supports many (but not all) of the native Python operations, producing corresponding (continuous) `Value` objects as results. These operations support the mixing of `Value` objects and other suitable Python objects. The following operators are supported:

- **Arithmetic**
  - `a + b`
  - `a - b`
  - `a * b`
  - `a @ b`
  - `a / b`
  - `a // b`
  - `a % b`
  - `divmod(a, b)`
  - `a ** b`
- **Bit-wise**
  - `a << b`
  - `a >> b`
  - `a & b`

  - `a | b`
  - `a ^ b`
- **Unary**
  - `-a`
  - `+a`
  - `abs(a)`
  - `~a`
- **Comparison**
  - `a < b`
  - `a <= b`
  - `a == b`
  - `a != b`
  - `a >= b`
  - `a > b`
- **Container operators**
  - `a[key]`
- **Numerical conversions**
  - `complex(a)`
  - `int(a)`
  - `float(a)`
  - `round(a)`
- **Python object/function usage**
  - `a(...)` will call the value as a function and return a `Value` containing the result. This value will be updated by re-calling the function whenever the Value changes. Like `fn()`, arguments may be `Value` objects and these will be unwrapped before the function is called and will also cause the function to be re-evaluated whenever they change. Do not use this to call functions with side effects.
  - `a.name` equivalent to `yarp.getattr(a, "name")`

Unfortunately this list *doesn't* include boolean operators (i.e. `not`, `and`, `or` and `bool`). This is due to a limitation of the Python data model which means that `bool` may only return an actual boolean value, not some other type of object. As a workaround you can substitute:

- `bool(a)` for `a == True` (works in most cases)
- `a and b` for `a & b` (works for boolean values but produces numbers)
- `a or b` for `a | b` (works for boolean values but produces numbers)

For a similar reasons, the `len` and `in` operators are also not supported.

This list also doesn't include mutating operators, for example `a[key] = b`. This is because the Python objects within a `Value` are treated as being immutable.

Finally, to reiterate, the result of these operators will always be continuous `Values`. For instantaneous versions of these operators, see the Python builtins section below.

3.5 Python builtins

The `yarp` API provides `Value`-compatible versions of a number of Python builtins and functions from the standard library:

- **Builtins**
  - `bool(a)`
  - `any(a)`
  - `all(a)`
  - `min(a)`
  - `max(a)`
  - `sum(a)`
  - `map(a)`
  - `sorted(a)`
  - `str(a)`
  - `repr(a)`
  - `str_format(a, ...)` (equivalent to `a.format(...)`)
  - `oct(a)`
  - `hex(a)`
  - `zip(a)`
  - `len(a)`
  - `getattr(object, name[, default])`
- Most non-mutating, non-underscore prefixed functions from the `operator` module.

These wrappers produce continuous `Value`s. Corresponding versions prefixed with `instantaneous_` are provided which produce instantaneous `Value`s.

3.6 Function wrappers

The primary mode of interaction with `yarp` `Value`s is intended to be via simple Python functions wrapped with `fn()` or `instantaneous_fn()`. These wrappers are defined below.

`yarp.`**`fn`**`(f)`

Decorator. Wraps a function so that it may be called with `Value` objects and itself return a persistent `Value`.

Say a function is defined and wrapped with `fn()` like so:

```
>>> @yarp.fn
... def add(a, b):
...     return a + b
```

The function can now be called with `Value` objects like so:

```
>>> a = yarp.Value(1)
>>> b = yarp.Value(2)
>>> c = add(a, b)
```

The returned value will itself be a `Value` object which will be updated whenever any of the arguments change.

```
>>> c.value
3
```

The wrapped function doesn't need to know anything about `Value` objects: the wrapper unpacks the `Value`s of each argument before passing it on and automatically wrapps the return value in a `Value`. (Non-`Value` arguments passed to the function are automatically passed through without modification).

The wrapped function is called once immediately when it is called and then again as required when its arguments change. The output `Value` will be persistent.

See also: *`instantaneous_fn()`*.

`yarp.`**`instantaneous_fn`**(*f*)

Decorator. Like `fn()` but the function output will be wrapped as an instantaneous `Value`.

The only other difference is that the function will not be called immediately and instead will only be called later when its inputs change.

3.7 General Value manipulation

The following utility functions are defined which accept and return `Value`s.

`yarp.`**`replace_novalue`**(*source_value*, *replacement_if_novalue=None*)

If the `source_value` is `NoValue`, return `replacement_if_novalue` instead.

Parameters

source_value [`Value`] An instantaneous or continuous `Value`.

replacement_if_novalue [Python object or `Value`] Replacement value to use if `source_value` has the value `NoValue`.

Returns

A continuous :py:class:'Value' which will be a copy of ''source_value'' if

''source_value'' is not :py:data:'NoValue', otherwise the value of

''replacement_if_novalue'' is used instead.

`yarp.`**`window`**(*source_value*, *num_values*)

Produce a moving window over a `Value`'s historical values.

This function treats the Value it is passed as a persistent Value, even if it is instantaneous (since a window function doesn't really have any meaning for a instantaneous values).

The `num_values` argument may be a (persistent) Value or a constant indicating the number of entries in the window. If this value later reduced, the contents of the window will be truncated immediately. If it is increaesd, any previously dropped values will not return. `num_values` is always assumed to be an integer greater than zero and never `NoValue`.

`yarp.`**`no_repeat`**(*source_value*)

Don't pass on change callbacks if the `Value` hasn't changed.

Works for both continuous and instantaneous `Value`s.

`yarp.`**`filter`**(*source_value*, *rule=NoValue*)

Filter change events.

The filter rule should be a function which takes the new value as an argument and returns a boolean indicating if the value should be passed on or not.

If the source value is persistent, the persistent value will remain unchanged when a value change is not passed on.

If the filter rule is `None`, non-truthy values and `NoValue` will be filtered out. If the filter rule is `NoValue` (the default) only `NoValue` will be filtered out.

3.8 Temporal Value manipulation

The following utility functions are defined which accept and return `Value`s but may delay or filter changes. These all use `asyncio` internally and require that a `asyncio.BaseEventLoop` be running.

`yarp.`**`delay`**(*source_value*, *delay_seconds*, *loop=None*)

Produce a time-delayed version of a `Value`.

Supports both instantaneous and continous `Values`. For continuous `Value`s, the initial value is set immediately.

The `delay_seconds` argument may be a constant or a Value giving the number of seconds to delay value changes. If it is increased, previously delayed values will be delayed further. If it is decreased, values which should already have been output will be output rapidly one after another.

The `loop` argument should be an `asyncio.BaseEventLoop` in which the delays will be scheduled. If `None`, the default loop is used.

`yarp.`**`time_window`**(*source_value*, *duration*, *loop=None*)

Produce a moving window over a `Value`'s historical values within a given time period.

This function treats the `Value` it is passed as a persistent `Value`, even if it is instantaneous (since a window function doesn't really have any meaning for an instantaneous value).

The `duration` may be a constant or a (persistent) Value giving the window duration as a number of seconds. The duration should be a number of seconds greater than zero and never be `NoValue`. If the value is reduced, previously inserted values will be expired earlier, possibly immediately if they should already have expired. If the value is increased, previously inserted values will have an increased timeout.

The `loop` argument should be an `asyncio.BaseEventLoop` in which windowing will be scheduled. If `None`, the default loop is used.

`yarp.`**`rate_limit`**(*source_value*, *min_interval=0.1*, *loop=None*)

Prevent changes occurring above a particular rate, dropping or postponing changes if necessary.

The `min_interval` argument may be a constant or a `Value`. If this value is decreased, currently delayed values will be output early (or immediately if the value would have been output previously). If increased, the current delay will be increased.

The `loop` argument should be an `asyncio.BaseEventLoop` in which the delays will be scheduled. If `None`, the default loop is used.

3.9 File-backed Values

The following function can be used to make *very* persistent `Value`s

`yarp.`**`file_backed_value`**(*filename*, *initial_value=NoValue*)

A persistent, file-backed value.

Upon creation, the value will be loaded from the specified filename. Whenever the value is changed it will be rewritten to disk. Changes made to the file while your program is running will be ignored.

If the file does not exist, it will be created and the value set to the value given by *initial_value*.

The value must be pickleable.

3.10 Time Values

The following function can be used to get the (continously changing) date and time:

`yarp.`**`now`**`(`*interval=1.0*, *tz=None*, *loop=None*`)`

Returns a continuous *`Value`* containing a `datetime.datetime` object holding the current time, refreshed every `interval` seconds.

The `interval` argument may be a constant or a *`Value`* giving the number of seconds to wait between updates. If the Value changes, the time until the next update will be reset starting from that moment in time.

The `tz` argument is passed on to `datetime.datetime.now()`. This must be a constant.

The `loop` argument should be an `asyncio.BaseEventLoop` in which the delays will be scheduled. If `None`, the default loop is used.

Python Module Index

Index